Sams Teach Yourself C++ in 24 Hours

Second Edition

Jesse Liberty

S/MS

A Division of Macmillan Computer Publishing
201 West 103rd St., Indianapolis, Indiana, 46290 USA

Disclaimer:
This netLibrary eBook does not include data from the CD-ROM that was part of the original hard copy book.

Sams Teach Yourself C++ in 24 Hours, Second Edition

Copyright © 1999 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-31516-5

Library of Congress Catalog Card Number: 98-89565

Printed in the United States of America

First Printing: February 1999

01 00 99   4 3 2

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The authors and the publisher shall have neither liability or responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the CD or programs accompanying it.

Executive Editor
Tracy Dunkelberger

Acquisitions Editor
Holly Allender

Development Editor
Sean Dixon

Managing Editor
Jodi Jensen

Senior Editor

Susan Ross Moore

Copy Editors
Mary Lagu
Linda Morris

Indexer
Mary Gammons

Proofreader
Eddie Lushbaugh

Technical Editors
Donald Xie
Sivaramakrishnan J.

Software Development Specialist
Dan Scherf

Team Coordinator
Michelle Newcomb

Interior Design
Gary Adair

Cover Design
Aren Howell

Layout Technicians
Ayanna Lacey
Heather Miller
Amy Parker

Hour 23
Templates

In the past few years, C++ has added a few new features. One of the more exciting and powerful new aspects of C++ is templates. Templates enable you to build type-safe collections. In this hour you will learn

· What templates are and how to use them

· Why templates supply a better alternative to macros

· How to create class templates

What Are Templates?

In Hour 19, you learned how to make a linked list. Your linked list was nicely encapsulated: the list knew only about its head pointer; the head pointer delegated its work to internal pointers, and so forth.

The one glaring problem with the linked list was that it only knew how to handle the particular data objects it was created to work with. If you wanted to put anything else into your linked list, you couldn't do it. You couldn't for <?>xample, make a linked list of Car objects, or of Cats, or of any other object <?>t wasn't of the same type as those in the original list.

To solve this problem, you could create a List base class and derive from it the CarList and CatsList classes. You could then cut and paste much of the LinkedList class into the new CatsList declaration. Next week, however, when you want to make a list of Car objects, you have to make a new class and 'cut and paste again.

Needless to say, this is not a satisfactory solution. Over time, the List class and its derived classes will have to be extended. Making sure that all the changes are propagated to all the related classes will then be a nightmare.

Templates offer a solution to this problem. In addition, unlike old-fashioned macros, templates are an integrated part of the language, type-safe, and very flexible.

Parameterized Types

Templates enable you to teach the compiler how to make a list of any type of thing, rather than creating a set of type-specific lists. A PartsList is a list of parts; a CatList is a list of cats. The only way in which they differ is the type of the thing on the list. With templates, the type of the thing on the list becomes a parameter to the definition of the class.

New Term: The act of creating a specific type from a template is called *instantiation*, and the individual classes are called *instances* of the template.

New Term: Templates provide you with the ability to create a general class and pass types as *parameters* to that class, to build specific instances.

Template Definition.

You declare a parameterized List object (a template for a list) by writing

```
1: template <class T>    // declare the template and the parameter
2: class List            // the class being parameterized
3: {
4: public:
5:     List();
6:     // full class declaration here
7: };
```

The keyword template is used at the beginning of every declaration and definition of a template class. The parameters of the template are after the keyword template; they are the items that will change with each instance. For example, in the list template shown in the previous code snippet, the type of the objects stored in the list will change. One instance might store an list of Integers, while another might store an list of Animals.

In this example, the keyword `class` is used, followed by the identifier T. The keyword `class` indicates that this parameter is a type. The identifier T is used throughout the rest of the template definition to refer to the parameterized type. One instance of this class will substitute `int` everywhere T appears, and another will substitute `Cat`.

To declare an `int` and a `Cat` instance of the parameterized list class, you would write

```
List<int> anIntList;
List<Cat> aCatList;
```

The object `anIntList` is of the type list of integers; the object `aCatList` is of the type `ListOfCats`. You can now use the type `List<int>` anywhere you would normally use a type—as the return value from a function, as a parameter to a function, and so forth.

Listing 23.1 parameterizes our `List` object. This is an excellent technique for building templates: Get your object working on a single type, as we did in Lesson 19, "Linked Lists." Then by parameterizing, generalize your object to handle any type.

LISTING **23.1** DEMONSTRATING PARAMETERIZED LISTS

```
1:     // **********************************************
2:     //     FILE:       Listing 23.1
3:     //
4:     //     PURPOSE:    Demonstrate parameterized list
5:     //     NOTES:
6:     //
7:     //  COPYRIGHT:   Copyright (C) 1997 Liberty Associates, Inc.
8:     //               All Rights Reserved
9:     //
10:    // Demonstrates an object-oriented approach to parameterized
11:    // linked lists. The list delegates to the node.
12:    // The node is an abstract Object type. Three types of
13:    // nodes are used, head nodes, tail nodes and internal
14:    // nodes. Only the internal nodes hold Object.
15:    //
16:    // The Object class is created to serve as an object to
17:    // hold in the linked list.
18:    //
19:    // **********************************************
20:
21:
22:    #include <iostream.h>
23:
24:    enum { kIsSmaller, kIsLarger, kIsSame};
25:
26:    // Object class to put into the linked list
27:    // Any class in this linked list must support two methods:
```

continues

LISTING 23.1 continued

```
28:     // Show (displays the value) and Compare / (returns relative pos
29:     class Data
30:     {
31:     public:
32:         Data(int val) :myValue(val){}
33:         ~Data()
34:         {
35:           cout << " Deleting Data object with value: " ;
36:           cout << myValue << " \n" ;
37:         }
38:         int Compare(const Data &);
39:         void Show() { cout << myValue << endl; }
40:     private:
41:         int myValue;
42:     };
43:
44:     // compare is used to decide where in the list
45:     // a particular object belongs.
46:     int Data::Compare(const Data & theOtherObject)
47:     {
48:         if (myValue < theOtherObject.myValue)
49:             return kIsSmaller;
50:         if (myValue > theOtherObject.myValue)
51:             return kIsLarger;
52:         else
53:             return kIsSame;
54:     }
55:
56:     // Another class to put into the linked list
57:     // Again, every class in this linked list must support // two me
58:     // Show (displays the value) and Compare // (returns relative pc
59:     class Cat
60:     {
61:     public:
62:         Cat(int age): myAge(age){}
63:         ~Cat()
64:         {
65:           cout << " Deleting " ;
66:           cout << myAge << " years old Cat.\n" ;
67:         }
68:         int Compare(const Cat &);
69:         void Show()
70:         {
71:           cout << " This cat is " ;
72:           cout << myAge << " years old\n" ;
73:         }
```

(

LISTING 23.1 continued

```
74:     private:
75:         int myAge;
76:     };
77:
78:
79:     // compare is used to decide where in the list
80:     // a particular object belongs.
81:     int Cat::Compare(const Cat & theOtherCat)
82:     {
83:         if (myAge < theOtherCat.myAge)
84:             return kIsSmaller;
85:         if (myAge > theOtherCat.myAge)
86:             return kIsLarger;
87:         else
88:             return kIsSame;
89:     }
90:
91:
92:     // ADT representing the node object in the list
93:     // Every derived class must override Insert and Show
94:     template <class T>
95:     class Node
96:     {
97:     public:
98:         Node(){}
99:         virtual ~Node(){}
100:        virtual Node * Insert(T * theObject)=0;
101:        virtual void Show() = 0;
102:    private:
103:    };
104:
105:    template <class T>
106:    class InternalNode: public Node<T>
107:    {
108:    public:
109:        InternalNode(T * theObject, Node<T> * next);
110:        ~InternalNode(){ delete myNext; delete myObject; }
111:        virtual Node<T> * Insert(T * theObject);
112:        virtual void Show()
113:        {
114:            myObject->Show();
115:            myNext->Show();
116:        } // delegate!
117:    private:
118:        T * myObject; // the Object itself
119:        Node<T> * myNext;  // points to next node in the linked list
120:    };
121:
122:    // All the constructor does is initialize
```

LISTING 23.1 continued

```
123:    template <class T>
124:    InternalNode<T>::InternalNode(T * theObject, Node<T> * next):
125:    myObject(theObject),myNext(next)
126:    {
127:    }
128:
129:    // the meat of the list
130:    // When you put a new object into the list
131:    // it is passed to the node which figures out
132:    // where it goes and inserts it into the list
133:    template <class T>
134:    Node<T> * InternalNode<T>::Insert(T * theObject)
135:    {
136:
137:        // is the new guy bigger or smaller than me?
138:        int result = myObject->Compare(*theObject);
139:
140:
141:        switch(result)
142:        {
143:            // by convention if it is the same as me it comes first
144:        case kIsSame:          // fall through
145:        case kIsLarger:        // new Object comes before me
146:            {
147:                InternalNode<T> * ObjectNode =
148:                new InternalNode<T>(theObject, this);
149:                return ObjectNode;
150:            }
151:
152:            // it is bigger than I am so pass it on to the next
153:            // node and let HIM handle it.
154:        case kIsSmaller:
155:                myNext = myNext->Insert(theObject);
156:                return this;
157:        }
158:        return this; // appease MSC
159:    }
160:
161:
162:    // Tail node is just a sentinel
163:    template <class T>
164:    class TailNode : public Node<T>
165:    {
166:    public:
167:        TailNode(){}
168:        virtual ~TailNode(){}
169:        virtual Node<T> * Insert(T * theObject);
170:        virtual void Show() { }
171:
```

continues

LISTING 23.1 continued

```
172:    private:
173:
174:    };
175:
176:    // If Object comes to me, it must be inserted before me
177:    // as I am the tail and NOTHING comes after me
178:    template <class T>
179:    Node<T> * TailNode<T>::Insert(T * theObject)
180:    {
181:        InternalNode<T> * ObjectNode =
182:        new InternalNode<T>(theObject, this);
183:        return ObjectNode;
184:    }
185:
186:    // Head node has no Object, it just points
187:    // to the very beginning of the list
188:    template <class T>
189:    class HeadNode : public Node<T>
190:    {
191:    public:
192:        HeadNode();
193:        virtual ~HeadNode() { delete myNext; }
194:        virtual Node<T> * Insert(T * theObject);
195:        virtual void Show() { myNext->Show(); }
196:    private:
197:        Node<T> * myNext;
198:    };
199:
200:    // As soon as the head is created
201:    // it creates the tail
202:    template <class T>
203:    HeadNode<T>::HeadNode()
204:    {
205:        myNext = new TailNode<T>;
206:    }
207:
208:    // Nothing comes before the head so just
209:    // pass the Object on to the next node
210:    template <class T>
211:    Node<T> * HeadNode<T>::Insert(T * theObject)
212:    {
213:        myNext = myNext->Insert(theObject);
214:        return this;
215:    }
216:
217:    // I get all the credit and do none of the work
218:    template <class T>
219:    class LinkedList
220:    {
```

LISTING **23.1** continued

```
221:    public:
222:        LinkedList();
223:        ~LinkedList() { delete myHead; }
224:        void Insert(T * theObject);
225:        void ShowAll() {myHead->Show(); }
226:    private:
227:        HeadNode<T> * myHead;
228:    };
229:
230:    // At birth, i create the head node
231:    // It creates the tail node
232:    // So an empty list points to the head which
233:    // points to the tail and has nothing between
234:    template <class T>
235:    LinkedList<T>::LinkedList()
236:    {
237:        myHead = new HeadNode<T>;
238:    }
239:
240:    // Delegate, delegate, delegate
241:    template <class T>
242:    void LinkedList<T>::Insert(T * pObject)
243:    {
244:        myHead->Insert(pObject);
245:    }
246:
247:    // test driver program
248:    int main()
249:    {
250:        Cat * pCat;
251:        Data * pData;
252:        int val;
253:        LinkedList<Cat>  ListOfCats;
254:        LinkedList<Data> ListOfData;
255:
256:        // ask the user to produce some values
257:        // put them in the list
258:        for (;;)
259:        {
260:            cout << " What value? (0 to stop): " ;
261:            cin >> val;
262:            if (!val)
263:                break;
264:            pCat = new Cat(val);
265:            pData= new Data(val);
266:            ListOfCats.Insert(pCat);
267:            ListOfData.Insert(pData);
268:        }
269:
```

LISTING 23.1 continued

```
270:              // now walk the list and show the Object
271:              cout << " \n" ;
272:              ListOfCats.ShowAll();
273:              cout << " \n" ;
274:              ListOfData.ShowAll();
275:              cout << " \n *********** \n\n" ;
276:              return 0; // The lists fall out of scope and // are destroye
277:        }
```

Output:

```
What value? (0 to stop): 5
What value? (0 to stop): 13
What value? (0 to stop): 2
What value? (0 to stop): 9
What value? (0 to stop): 7
What value? (0 to stop): 0


This cat is 2 years old
This cat is 5 years old
This cat is 7 years old
This cat is 9 years old
This cat is 13 years old


2
5
7
9
13


***********

Deleting Data object with value: 13
Deleting Data object with value: 9
Deleting Data object with value: 7
Deleting Data object with value: 5
Deleting Data object with value: 2
Deleting 13 years old Cat.
Deleting 9 years old Cat.
Deleting 7 years old Cat.
Deleting 5 years old Cat.
Deleting 2 years old Cat.
```

Analysis: The first thing to notice is the striking similarity to the listing in Hour 19. Go ahead, find the original listing; I'll wait right here…. As you can see, little has changed.

The biggest change is that each of the class declarations and methods is prepended with

```
template class <T>
```

This tells the compiler that you are parameterizing this list on a type that you will define later, when you instantiate the list. For example, the declaration of the Node class now becomes

```
template <class T>
class Node
```

This indicates that Node will not exist as a class in itself, but rather that you will instantiate Nodes of Cats and Nodes of Data objects. The actual type you'll pass in is represented by T.

Thus, InternalNode now becomes InternalNode<T> (read that as "InternalNode of T"). And InternalNode<T> points not to a Data object and another Node; rather, it points to a T (whatever type of object) and a Node<T>. You can see this on lines 118 and 119.

Look carefully at Insert, defined on lines 133–159. The logic is just the same, but where we used to have a specific type (Data) we now have T. Thus, on line 134 the parameter is a pointer to a T. Later, when we instantiate the specific lists, the T will be replaced by the compiler with the right type (Data or Cat).

The important thing is that the InternalNode can continue working, indifferent to the actual type. It knows to ask the objects to compare themselves. It doesn't care whether Cats compare themselves in the same way Data objects do. In fact, we can rewrite this so that Cats don't keep their age; we can have them keep their birth date and compute their relative age on the fly, and the InternalNode won't care a bit.

Using Template Items

You can treat template items as you would any other type. You can pass them as parameters, either by reference or by value, and you can return them as the return values of functions, also by value or by reference. Listing 23.2 demonstrates how to pass Template objects.

LISTING 23.2 DEMONSTRATING PARAMETERIZED LISTS

```
1:      // **************************************************
2:      //      FILE:        Listing 23.2
3:      //
4:      //      PURPOSE:     Demonstrate parameterized list
5:      //      NOTES:
6:      //
7:      // COPYRIGHT:  Copyright (C) 1997 Liberty Associates, Inc.
8:      //                  All Rights Reserved
9:      //
10:     // Demonstrates an object-oriented approach to parameterized
```

continues

LISTING 23.2 continued

```
11:        // linked lists. The list delegates to the node.
12:        // The node is an abstract Object type. Three types of
13:        // nodes are used, head nodes, tail nodes and internal
14:        // nodes. Only the internal nodes hold Object.
15:        //
16:        // The Object class is created to serve as an object to
17:        // hold in the linked list.
18:        //
19:        // **************************************************
20:
21:
22:        #include <iostream.h>
23:
24:
25:        enum { kIsSmaller, kIsLarger, kIsSame};
26:
27:        // Object class to put into the linked list
28:        // Any class in this linked list must support two methods:
29:        // Show (displays the value) and Compare // (returns relative p
30:        class Data
31:        {
32:        public:
33:            Data(int val):myValue(val){}
34:            ~Data()
35:            {
36:                cout << " Deleting Data object with value: " ;
37:                cout << myValue << " \n" ;
38:            }
39:            int Compare(const Data &);
40:            void Show() { cout << myValue << endl; }
41:        private:
42:            int myValue;
43:        };
44:
45:        // compare is used to decide where in the list
46:        // a particular object belongs.
47:        int Data::Compare(const Data & theOtherObject)
48:        {
49:            if (myValue < theOtherObject.myValue)
50:                return kIsSmaller;
51:            if (myValue > theOtherObject.myValue)
52:                return kIsLarger;
53:            else
54:                return kIsSame;
55:        }
56:
57:        // Another class to put into the linked list
```

c

LISTING 23.2 continued

```
58:        // Again, every class in this linked list must support // two m
59:        // Show (displays the value) and Compare // (returns relative p
60:        class Cat
61:        {
62:        public:
63:            Cat(int age): myAge(age){}
64:            ~Cat(){cout << " Deleting " << myAge << " years old Cat.\n"
65:            int Compare(const Cat &);
66:            void Show() { cout << " This cat is " << myAge << " years ol
67:        private:
68:            int myAge;
69:        };
70:
71:
72:        // compare is used to decide where in the list
73:        // a particular object belongs.
74:        int Cat::Compare(const Cat & theOtherCat)
75:        {
76:            if (myAge < theOtherCat.myAge)
77:                return kIsSmaller;
78:            if (myAge > theOtherCat.myAge)
79:                return kIsLarger;
80:            else
81:                return kIsSame;
82:        }
83:
84:
85:        // ADT representing the node object in the list
86:        // Every derived class must override Insert and Show
87:        template <class T>
88:        class Node
89:        {
90:        public:
91:            Node(){}
92:            virtual ~Node(){}
93:            virtual Node * Insert(T * theObject)=0;
94:            virtual void Show() = 0;
95:        private:
96:        };
97:
98:        template <class T>
99:        class InternalNode: public Node<T>
100:       {
101:       public:
102:           InternalNode(T * theObject, Node<T> * next);
```

LISTING **23.2** continued

```
103:        virtual ~InternalNode(){ delete myNext; delete myObject; }
104:        virtual Node<T> * Insert(T * theObject);
105:        virtual void Show() // delegate!
106:        {
107:            myObject->Show(); myNext->Show();
108:        }
109:    private:
110:        T * myObject;  // the Object itself
111:        Node<T> * myNext; // points to next node in the linked list
112:    };
113:
114:    // All the constructor does is initialize
115:    template <class T>
116:    InternalNode<T>::InternalNode(T * theObject, Node<T> * next):
117:    myObject(theObject),myNext(next)
118:    {
119:    }
120:
121:    // the meat of the list
122:    // When you put a new object into the list
123:    // it is passed to the node which figures out
124:    // where it goes and inserts it into the list
125:    template <class T>
126:    Node<T> * InternalNode<T>::Insert(T * theObject)
127:    {
128:
129:        // is the new guy bigger or smaller than me?
130:        int result = myObject->Compare(*theObject);
131:
132:
133:        switch(result)
134:        {
135:        // by convention if it is the same as me it comes first
136:        case kIsSame:        // fall through
137:        case kIsLarger:     // new Object comes before me
138:            {
139:                InternalNode<T> * ObjectNode =
140:                new InternalNode<T>(theObject, this);
141:                return ObjectNode;
142:            }
143:
144:            // it is bigger than I am so pass it on to the next
145:            // node and let HIM handle it.
146:         case kIsSmaller:
147:                myNext = myNext->Insert(theObject);
148:                return this;
149:         }
150:        return this;  // appease MSC
```

continues

Listing 23.2 continued

```
151:    }
152:
153:
154:    // Tail node is just a sentinel
155:    template <class T>
156:    class TailNode : public Node<T>
157:    {
158:    public:
159:        TailNode(){}
160:        virtual ~TailNode(){}
161:        virtual Node<T> * Insert(T * theObject);
162:        virtual void Show() {}
163:
164:    private:
165:
166:    };
167:
168:    // If Object comes to me, it must be inserted before me
169:    // as I am the tail and NOTHING comes after me
170:    template <class T>
171:    Node<T> * TailNode<T>::Insert(T * theObject)
172:    {
173:        InternalNode<T> * ObjectNode =
174:            new InternalNode<T>(theObject, this);
175:        return ObjectNode;
176:    }
177:
178:    // Head node has no Object, it just points
179:    // to the very beginning of the list
180:    template <class T>
181:    class HeadNode : public Node<T>
182:    {
183:    public:
184:        HeadNode();
185:        virtual ~HeadNode() { delete myNext; }
186:        virtual Node<T> * Insert(T * theObject);
187:        virtual void Show() { myNext->Show(); }
188:    private:
189:        Node<T> * myNext;
190:    };
191:
192:    // As soon as the head is created
193:    // it creates the tail
194:    template <class T>
195:    HeadNode<T>::HeadNode()
196:    {
197:        myNext = new TailNode<T>;
198:    }
199:
```

LISTING **23.2** continued

```
200:     // Nothing comes before the head so just
201:     // pass the Object on to the next node
202:     template <class T>
203:     Node<T> * HeadNode<T>::Insert(T * theObject)
204:     {
205:         myNext = myNext->Insert(theObject);
206:         return this;
207:     }
208:
209:     // I get all the credit and do none of the work
210:     template <class T>
211:     class LinkedList
212:     {
213:     public:
214:         LinkedList();
215:         ~LinkedList() { delete myHead; }
216:         void Insert(T * theObject);
217:         void ShowAll() { myHead->Show(); }
218:     private:
219:         HeadNode<T> * myHead;
220:     };
221:
222:     // At birth, i create the head node
223:     // It creates the tail node
224:     // So an empty list points to the head which
225:     // points to the tail and has nothing between
226:     template <class T>
227:     LinkedList<T>::LinkedList()
228:     {
229:         myHead = new HeadNode<T>;
230:     }
231:
232:     // Delegate, delegate, delegate
233:     template <class T>
234:     void LinkedList<T>::Insert(T * pObject)
235:     {
236:         myHead->Insert(pObject);
237:     }
238:
239:     void myFunction(LinkedList<Cat>& ListOfCats);
240:     void myOtherFunction(LinkedList<Data>& ListOfData);
241:
242:     // test driver program
243:     int main()
244:     {
245:         LinkedList<Cat>  ListOfCats;
246:         LinkedList<Data> ListOfData;
247:
248:         myFunction(ListOfCats);
```

LISTING 23.2 continued

```
249:            myOtherFunction(ListOfData);
250:
251:            // now walk the list and show the Object
252:            cout << " \n" ;
253:            ListOfCats.ShowAll();
254:            cout << " \n" ;
255:            ListOfData.ShowAll();
256:            cout << " \n *********** \n\n" ;
257:            return 0;  // The lists fall out of scope and are // destroye
258:        }
259:
260:    void myFunction(LinkedList<Cat>& ListOfCats)
261:        {
262:            Cat * pCat;
263:            int val;
264:
265:            // ask the user to produce some values
266:            // put them in the list
267:            for (;;)
268:            {
269:                cout << " \nHow old is your cat? (0 to stop): " ;
270:                cin >> val;
271:                if (!val)
272:                    break;
273:                pCat = new Cat(val);
274:                ListOfCats.Insert(pCat);
275:            }
276:
277:        }
278:
279:    void myOtherFunction(LinkedList<Data>& ListOfData)
280:        {
281:            Data * pData;
282:            int val;
283:
284:            // ask the user to produce some values
285:            // put them in the list
286:            for (;;)
287:            {
288:                cout << " \nWhat value? (0 to stop): " ;
289:                cin >> val;
290:                if (!val)
291:                    break;
292:                pData = new Data(val);
293:                ListOfData.Insert(pData);
294:            }
295:
296:        }
```

Output:

```
How old is your cat? (0 to stop): 12
How old is your cat? (0 to stop): 2
How old is your cat? (0 to stop): 14
How old is your cat? (0 to stop): 6
How old is your cat? (0 to stop): 0
What value? (0 to stop): 3
What value? (0 to stop): 9
What value? (0 to stop): 1
What value? (0 to stop): 5
What value? (0 to stop): 0
This cat is 2 years old
This cat is 6 years old
This cat is 12 years old
This cat is 14 years old


1
3
5
9


* * * * * * * * * * *


Deleting Data object with value: 9
Deleting Data object with value: 5
Deleting Data object with value: 3
Deleting Data object with value: 1
Deleting 14 years old Cat.
Deleting 12 years old Cat.
Deleting 6 years old Cat.
Deleting 2 years old Cat.
```

Analysis: This code is much like the previous example, but this time we pass the `LinkedLists` by reference to their respective functions for processing. This is a powerful feature. After the lists are instantiated, they can be treated as fully defined types, passed into functions, and returned as values.

Page 420

The Standard Template Library

A new development in C++ is the adoption of the *Standard Template Library* (STL). All the major compiler vendors now offer the STL as part of their compiler. STL is a library of template-based container classes, including vectors, lists, queues, and stacks. The STL also includes a number of common algorithms, including sorting and searching.

The goal of the STL is to give you an alternative to reinventing the wheel for these common requirements. The STL is tested and debugged, offers high performance, and its free! Most important, the STL is reusable; when you understand how to use an STL container, you can use it in all your programs without reinventing it.

Summary

In this hour you learned how to create and use templates. Templates are a built-in facility of C++ used to create parameterized types—types that change their behavior based on parameters passed in at creation. They are a way to reuse code safely and effectively.

The definition of the template determines the parameterized type. Each instance of the template is an actual object, which can be used like any other object—as a parameter to a function, as a return value, and so forth.

Q&A

Q Why use templates when macros will do?

A Templates are type-safe and built into the language.

Q What is the difference between the parameterized type of a template function and the parameters to a normal function?

A A regular function (non-template) takes parameters on which it may take action. A template function allows you to parameterize the type of a particular parameter to the function. That is, you can pass an ListOfType to a function, and then have the *Type* determined by the template instance.

Q When do you use templates and when do you use inheritance?

A Use templates when all the behavior or virtually all the behavior is unchanged, but the type of the item on which your class acts is different. If you find yourself copying a class and changing only the type of one or more of its members, it may be time to consider using a template.